REMARKS

The Office Action indicates that claims 1-39 are pending, but claims 4-13, 17-26 and 30-39 were canceled in the Preliminary Amendment filed July 23, 2003. Consequently, only claims 1-3, 14-16 and 27-29 are pending.

Claims 1-3, 14-16 and 27-29 stand rejected under 35 U.S.C. § 102(e) as being anticipated by U.S. Patent No. 5,539,907 (Srivastava). The applicants submit that several features of the claims are not taught by Srivastava. Before addressing specific claim recitations that are missing from Srivastava, it is helpful to understand a fundamental difference between the claimed invention and that of Srivastava.

Generally, the claimed invention is directed to a method of translating a computer program compiled to run on a first processor (*i.e.,* computer) that implements a first instruction set so that it can run on a second processor that implements a different instruction set. Specifically, the compiled machine code corresponding to the instruction set of the first processor (*i.e.,* a first compiled code state) is translated to compiled machine code corresponding to the instruction set of a second, different processor (*i.e.,* a second compiled code state).[1]

On the contrary, the system described in Srivastava has nothing to do with translating a computer program compiled to run on one processor instruction set into compiled machine code for a different processor instruction set. Rather, Srivastava describes a method that allows a developer to modify a computer program to insert certain performance monitoring code that will allow the developer to monitor the performance of the computer program when it runs. Specifically, in order to aid in monitoring the performance of a computer program, the object code modules of a computer program are "translated," *not* to machine code for a different processor instruction set, but to "a single linked code module in the form of a *machine independent* register transfer language."[2] In this intermediate form, the developer can examine the code flow and then more easily modify the program to add performance monitoring procedures. Specifically, an "organizer 54 partitions the program into basic structural components including ... basic blocks" and then builds a procedure flow graph

---

[1] *See,* Spec., p. 1, ln. 15 – p. 2, ln. 5; and claims 1, 14 and 27.
[2] Srivastava, col. 2, ll. 41-43 (emphasis added).

(PFG) 200 and a program call graph (PCG) 300 to enable the execution flow of the program to be traced.[3] A user then uses an "instrumentor 55" to identify and modify specific portions of the program to be monitored.[4] Once the user finishes modifying the program to add performance monitoring code, a "code generator 57" is used to generate machine-dependent code for the hardware architecture on which the program runs.[5]

Thus, the claimed invention involves translating a compiled computer program from a first compiled code state (corresponding to a first instruction set) to a second compiled code state (corresponding to a second instruction set), whereas the system of Srivastava translates compiled modules of a program to an intermediate form to enable a developer to modify the program to add performance monitoring code before generating the executable machine code for the program. Given the fundamental differences between the two systems, it is not surprising that a number of features of the claims are not found in the Srivastava reference.

First, independent claims 1, 14 and 27 each expressly recite "translating compiled programming code from a first compiled code state to a second compiled code state." As explained above and in the specification, the terms "first compiled code state" and "second compiled code state" refer to the corresponding instruction sets from/to which the code is being translated.[6] Srivastava does not disclose translating the compiled programming code of a computer program from a first compiled code state to a second compiled code state. Rather, Srivastava discloses a system in which the compiled modules of a computer program are translated to an intermediate form to enable the computer program to be modified before generating machine code for the processor on which the program will run. Because Srivastava does not disclose "translating compiled programming code from a first compiled code state to a second compiled code state," as recited in claims 1, 14 and 27, it does not anticipate those claims.

Second, claims 1, 14 and 27 each further recite:

> exploring, based on the CFG, ... all identified basic blocks that
> lead to [a] *dynamic branch* as far back as is necessary *to fully*

---

[3] Srivastava, col. 4, ln. 66 – col. 5, ln. 16.
[4] Srivastava, col. 5, ll. 17-23.
[5] Srivastava, col. 5, ll. 24-33.
[6] *Spec.*, p. 7, ll. 24-26.

> *determine a set of destination addresses for the dynamic*
> *branch . . . .*

Srivastava also does not teach this feature.

As recited earlier in each claim (and as defined in the specification), a "dynamic branch" is "a transfer to one of a set of destinations based on a calculation of a destination address."[7] That is, the branch is "dynamic" because the destination address is not known ahead of time, but instead is calculated as the program executes and thus may be one of many possible destinations. The Office Action cites to col. 4:1-10 of Srivastava ("...by monitoring *conditional* branch instructions at the end of basic blocks") (emphasis added) as teaching the concept of a dynamic branch, but a conditional branch is not a dynamic branch. A conditional branch is a branch in which the *decision to branch or not* is based on some condition or calculation, as opposed to a dynamic branch where it is the *destination address* of the branch that is calculated. Indeed, a dynamic branch can be unconditional or conditional.[8] While the cited portion of Srivastava mentions conditional branches, there does not appear to be any mention of "dynamic" branches in Srivastava.

Moreover, all that Srivastava appears to teach is "locating all conditional branch instruction in [a] program." Col. 11, ll. 14-20. There does not appear to be any description of "exploring ... all identified basic blocks that lead to [a] dynamic branch as far back as is necessary to fully determine a set of destination addresses for the dynamic branch," as recited in the claims. The Office Action refers to col. 6:35-53 ("...reveals all possible execution destinations") as teaching this feature, but that portion of Srivastava is discussing how "case-statements" are handled when generating control graphs. The full paragraph is:

> By identifying all of the case-statements in a program, the jump
> table can be partitioned into a set of branch tables of a known
> size. This in turn *reveals all possible execution destinations.*
> The execution destinations can be used to create the control
> graphs 200 and 300.

---

[7] *See*, claims 1, 14 and 27; *see also*, Spec., p. 2, ll. 11-16.
[8] *See*, Spec., p. 5, ll. 1-2 ("E-Mode has three dynamic branch instructions, one *unconditional* (DBUN), the other two *conditional* (DBFL & DBTR).")(emphasis added).

Clearly, this has nothing to do with "exploring, based on [a control flow graph] ... all identified basic blocks that lead to [a] dynamic branch as far back as is necessary to fully determine a set of destination addresses for the dynamic branch," as claimed by applicants. Thus, this feature is not taught by Srivastava, and this is another reason that Srivastava does not anticipate claims 1, 14 and 27.

Third, each of claims 1, 14 and 27 recites "translating the programming code from the first compiled code state to the second compiled code state based at least in part on the updated CFG." Thus, as claimed, the translation is from a "first *compiled* code state" to a "second *compiled* code state," and the translation is performed "based at least in part on the updated CFG." The Office Action cites to col. 5:37-50 of Srivastava ("...The translator converts the program into a linked module...") as teaching this feature. However, the "translator" of Srivastava does not translate from one compiled state to another. Rather, Srivastava's translator "transforms the program in object module form ... to a single linked code module ... in the form of an intermediate machine independent register transfer language (RTL)."[9] Moreover, that translation occurs before Srivastava's "organizer" creates the PCG and PFG control graphs, and therefore, that translation does not appear to be "based at least in part on" any control flow graph – contrary to claims 1, 14 and 17. Again, therefore, the cited portion of Srivastava does not anticipate the claimed feature.

Thus, as demonstrated above, numerous features of claims 1, 14 and 27 are missing from the teachings of Srivastava and, therefore, Srivastava does not anticipate those claims. Reconsideration of the Section 102(e) rejection of claims 1, 14 and 27 is therefore respectfully requested. Inasmuch as claims 2-3, 15-16 and 28-29 depend either directly or indirectly from one of those independent claims, the applicants submit that they too are patentable for the same reasons.
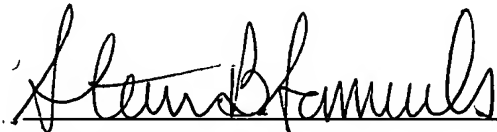
---

[9] Srivastava, col. 4, ll. 45-54.

CONCLUSION

For all the foregoing reasons, the applicants respectfully submit that the present application is in condition for allowance. An early Notice of Allowance is respectfully requested.

Date: June 18, 2007

Steven B. Samuels
Registration No. 37,711

Woodcock Washburn LLP
Cira Centre
2929 Arch Street, 12th Floor
Philadelphia, PA 19104-2891
Telephone: (215) 568-3100
Facsimile: (215) 568-3439

Correspondence address:
Richard J. Gregson, Esq.
Unisys Corporation
Unisys Way
Blue Bell, Pennsylvania 19424-0001